Praxisprojekt

ENTWICKLUNG EINER OSGI-BASIERTEN ANDROID-SOFTWARE ZUM STEUERN VON GERÄTEN IM BEREICH VON AMBIENT ASSISTED LIVING

FH-Düsseldorf

Verfasser:	Andreas Knoll	Matrikel-Nr.: 580537
1. Prüfer:	Prof. Dr. rer. nat. Lux	
2. Prüfer:	Prof. Dr.-Ing. Schaarschmidt	
Eingereicht am:	26.04.2013	

Abstract

This project deals with the development of an Android application which can be used as a control station for devices and sensors in an ambient assisted living platform. Bidirectional communication between a smartphone / tablet and the AAL devices is realized through an OSGi framework which permanently runs in the background.

This paper gives an insight into programming for Android, describes the customization of an OSGi framework to match the latest software design guidelines and gives a detailed documentation of the conception and realization of an application that meets the requirements

Zusammenfassung

Dieses Projekt befasst sich mit der Entwicklung einer Android-Software zur Steuerung von Geräten und Sensoren im Umfeld einer Ambient Assisted Living – Plattform. Ein durchgehend im Hintergrund betriebenes OSGi-Framework ermöglicht dabei die bidirektionale Kommunikation zwischen dem Smartphone / Tablet und den AAL-Geräten.

Die Projektarbeit gibt einen Einblick in die Android-Programmierung, beschreibt die Anpassung eines OSGi-Frameworks an neue Software-Design-Richtlinien und dokumentiert detailliert die Konzeption und Realisierung einer Anwendung, welche die gewünschten Anforderungen erfüllt.

Inhaltsverzeichnis

Inhalt

Inhaltsverzeichnis

1 Einleitung

Ambient Assisted Living (AAL) umfasst Methoden, Produkte und Dienstleistungen, welche es älteren und körperlich benachteiligten Menschen ermöglichen, möglichst lange in ihrem gewohnten Umfeld zu leben.

Das Labor für Informatik an der FH-Düsseldorf ist seit längerer Zeit in diesem Bereich tätig. Im Rahmen des WieDAS[1]-Projekts [1] wurden verschiedene Geräte (z.B. Wassermelder, Temperatursensor etc.) und Software zur Steuerung von diesen entwickelt. Das System sollte von einem Windows-Rechner sowie über ein Smartphone und Tablet bedienbar sein. Aus diesem Grund haben sich bereits die ehemaligen Studierenden Thomas Schmitz und Niels Wientzek in ihren Praxisprojekten und Bachelor-Thesen damit beschäftigt, Software zum Zugriff auf die Geräte für ein Android-Smartphone zu entwickeln.

In dem Zusammenhang entstanden Lösungen für ein unter Android lauffähiges OSGi-Framework sowie prototypische Implementierungen für einige Geräte. Die Dokumentation zu diesen Projekten kann in Thomas Schmitz‘ Praxisprojekt [2] respektive Niels Wientzeks Bachelor-Thesis [3] nachgelesen werden.

Ziel dieses Praxisprojekts ist der Entwurf und Realisierung einer einzigen Android-Anwendung zum Zugriff auf alle Geräte des WieDAS-Systems. Die Funktionalität soll folgende Punkte umfassen:

- Zugriff auf den Status des Geräts
- Audio-visuelles Feedback bei Alarm
- Bidirektionale Kommunikation zwischen Gerät und Smartphone

Dieses Dokument soll einen Überblick über die allgemeine Android-Programmierung geben und den Lösungsansatz für den konkreten Anwendungsfall präsentieren.

[1] Wiesbaden-Düsseldorfer Ambient Assisted Living Service Plattform

„[Die GUI] muss durch Testen unter verschiedenen Bildschirmauflösungen […] so erstellt werden, dass sie auf den meisten Geräten angenehm zu bedienen ist und den Design-Vorstellungen des Entwicklers gerecht wird.“

2 Einführung in die Android-Programmierung

2.1 Was ist Android?

Android ist ein auf dem Linux-Kernel basierendes mobiles Betriebssystem, das von der Open Handset Alliance – dessen Hauptmitglied Google ist – entwickelt wird.

Wie man es von einem Linux-artigen System erwarten würde, ist Android quelloffen und höchstgradig anpassbar. So haben Gerätehersteller und Softwareentwickler die Möglichkeit, das System an ihr Produkt respektive einen konkreten Anwendungsfall anzupassen. Was einerseits ein Vorteil ist, stellt den Entwickler andererseits oft vor Probleme.

So gibt es eine Vielzahl von unterschiedlichen Geräten, die sich alle in Bildschirmgröße und Auflösung unterscheiden, was das Erstellen einer grafischen Benutzeroberfläche erschwert. Diese muss durch Testen unter verschiedenen Bildschirmauflösungen – zum Beispiel innerhalb eines Emulators in der Entwicklungsumgebung – so erstellt werden, dass sie auf den meisten Geräten angenehm zu bedienen ist und den Design-Vorstellungen des Entwicklers gerecht wird.

Ein weiteres Problem stellt die Verzeichnisstruktur dar. Absolute Pfadangaben sind nur bedingt nutzbar, da einige Hersteller sich nicht an die Vorgabe von Google halten und ihre eigene Struktur anlegen. Dabei kann es sogar herstellerintern von einer Geräteserie zur anderen Unterschiede geben.

Bei der Anwendungsentwicklung für Android gibt es also einiges zu beachten. Die Problematik der unterschiedlichen Geräte und Android-Versionen wird im Verlauf dieser Projektarbeit immer wieder auftauchen und Lösungen werden vorgestellt.

2.2 Systemarchitektur

Um die Arbeit mit einem System zu erleichtern, ist es wichtig, den Aufbau von diesem zu verstehen. Die Abbildung 1 zeigt den Systemaufbau und benennt die einzelnen Schichten der Architektur.

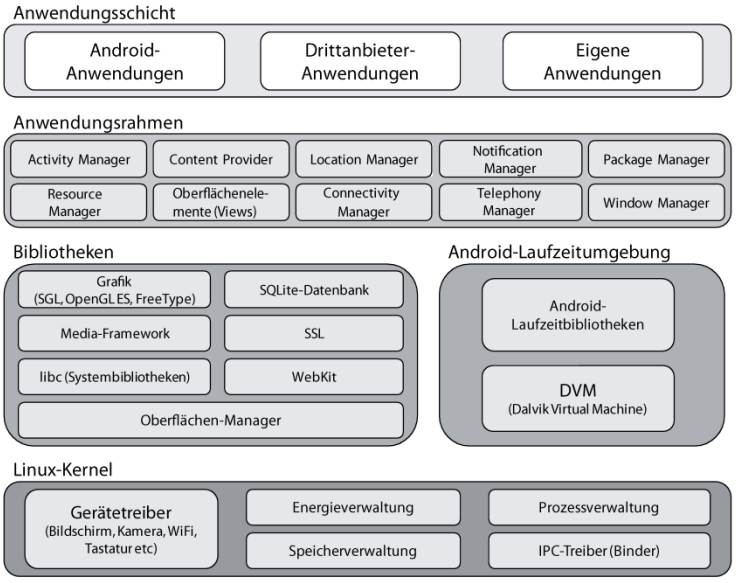

Abbildung 1: Android-Architektur [3]

An dieser Stelle sollen die einzelnen Schichten kurz vorgestellt und erläutert werden.

Linux-Kernel

Die Basis von Android stellt ein angepasster Linux-Kernel dar. Anfangs wurde die Version 2.6 verwendet – seit Android 4.0 wird Kernel 3.x benutzt. Der Linux-Kernel stellt Basisdienste zur Verfügung (Speicherverwaltung, Treiber, Prozessverwaltung etc.) und dient der Abstraktion der Hardware. Für Android wurde dieser an die Bedürfnisse mobiler Endgeräte angepasst. Die Optimierungen sorgen für weniger Energiebedarf und ressourcenschonende Interprozesskommunikation durch Verwendung eines speziellen IPC-Treibers (Binder), welcher die IPC als Shared-Memory-Zugriffe realisiert.

2 Einführung in die Android-Programmierung

Android Laufzeitumgebung

Die zwei Komponenten der Laufzeitumgebung sind:

- (angepasste) Java Laufzeit-Bibliotheken
- Dalvik Virtual Machine

Interessant ist hier vor allem die Dalvik VM. Diese basiert auf der quelloffenen Java VM Apache Harmony und arbeitet mit einem eigenen Bytecode. Der Einsatz einer eigenen virtuellen Maschine hat den Vorteil, dass weitere Android-spezifische Optimierungen vorgenommen werden können. Die Dalvik VM arbeitet mit einer Register-Maschine und ist darauf ausgelegt, mehrere Instanzen parallel laufen lassen zu können, was deutliche Geschwindigkeitsvorteile bei der Ausführung von Android-Anwendungen bringt.

Bibliotheken

Wie bei Linux-Distributionen, findet man in einem Android-System viele Standard C/C++ Bibliotheken. Auch sie sind meist für den Einsatz auf einem mobilen System optimiert. Zu den vorhandenen Bibliotheken zählen unter anderem die Codes für Multimedia und Daten, das WebKit für den Browser sowie die für den Linux-Kernel benötigte C-Bibliothek *libc*.

Anwendungsrahmen

Der Anwendungsrahmen (Application Framework) abstrahiert die Hardware und stellt dem Entwickler Komponenten für die GUI-Entwicklung zur Verfügung. Dabei stehen Android-spezifische APIs zur Verfügung, welche Zugriff auf alle Funktionen des Betriebssystems gewähren.

Anwendungsschicht

In der Anwendungsschicht befinden sich die eigentlichen Programme (*Apps*). Darunter sind nicht nur die Basisanwendungen, sondern auch vom Gerätehersteller vorinstallierte Software sowie Apps von Drittanbietern.

„Der Anwendungslebenszyklus beschreibt die einzelnen Zustände einer Android-Anwendung vom ersten Aufruf über Pausierung bis hin zum Beenden der App.“

2.3 Anwendungslebenszyklus

Der Anwendungslebenszyklus beschreibt die einzelnen Zustände einer Android-Anwendung vom ersten Aufruf über Pausierung bis hin zum Beenden der App. Dabei wird zwischen dem Lebenszyklus von Activities (Klassen, welche zur Darstellung der GUI benutzt werden) und Fragments (Modulare Teile einer Activity) unterschieden. Zwischen den Zuständen werden vom Betriebssystem bestimmte Methoden aufgerufen, welche der Entwickler überschreiben und somit auf Änderungen reagieren kann.

Beim Start einer Anwendung wird die *onCreate*-Methode der Activity ausgeführt, gefolgt von den Methoden *onStart* und *onResume*. Innerhalb dieses Ablaufs sollten statische Variablen definiert, die Oberfläche geladen und Netzwerkverbindungen gestartet werden. Danach befindet sich die Anwendung im Vordergrund.

Wird nun eine neue Anwendung oder Activity aufgerufen, so wird zunächst die *onPause*-Methode und in dem Fall, dass die Activity nicht mehr im Vordergrund ist, die *onStop*-Methode abgearbeitet. Danach verbleibt die Anwendung oder Activity in diesem Zustand bis sie vom Nutzer wieder aufgerufen oder vom System beendet wird.

Wird die Anwendung vom Nutzer wieder aufgerufen, so werden die Methoden *onRestart*, *onStart* und *onResume* durchlaufen. Beendet dieser die Anwendung, dann wird als letztes die *onDestroy*-Methode aufgerufen.

Fragments existieren nur im Kontext einer Activity und müssen sich somit dem Activity-Lebenszyklus unterordnen. Allerdings besitzen Fragments auch einen eigenen Lebenszyklus, der es ermöglicht, diese in unterschiedliche Zustände zu versetzen, ohne dass die darüber liegende Activity beeinflusst wird.

Eine grafische Darstellung der spezifischen Lebenszyklen liefern die Abbildungen 2 und 3.

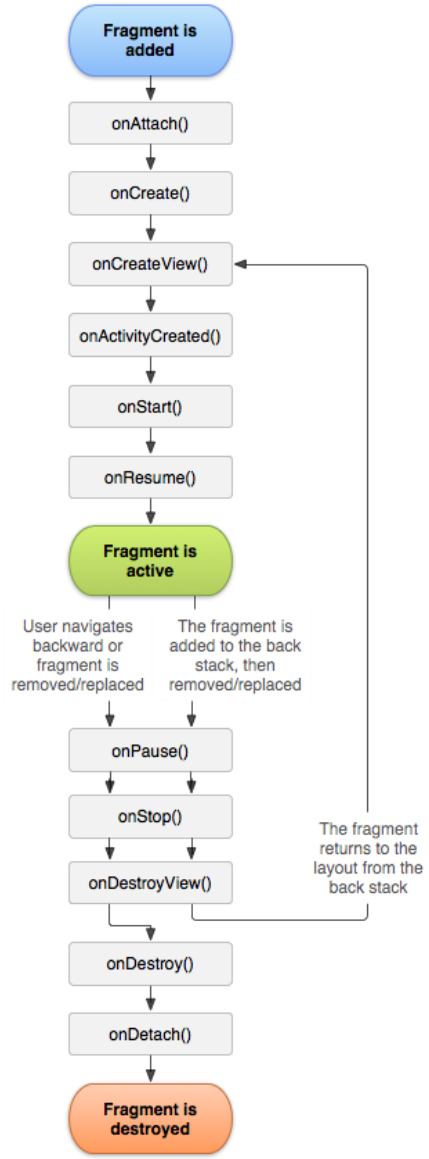

Abbildung 2: Fragment-Lebenszyklus [4]

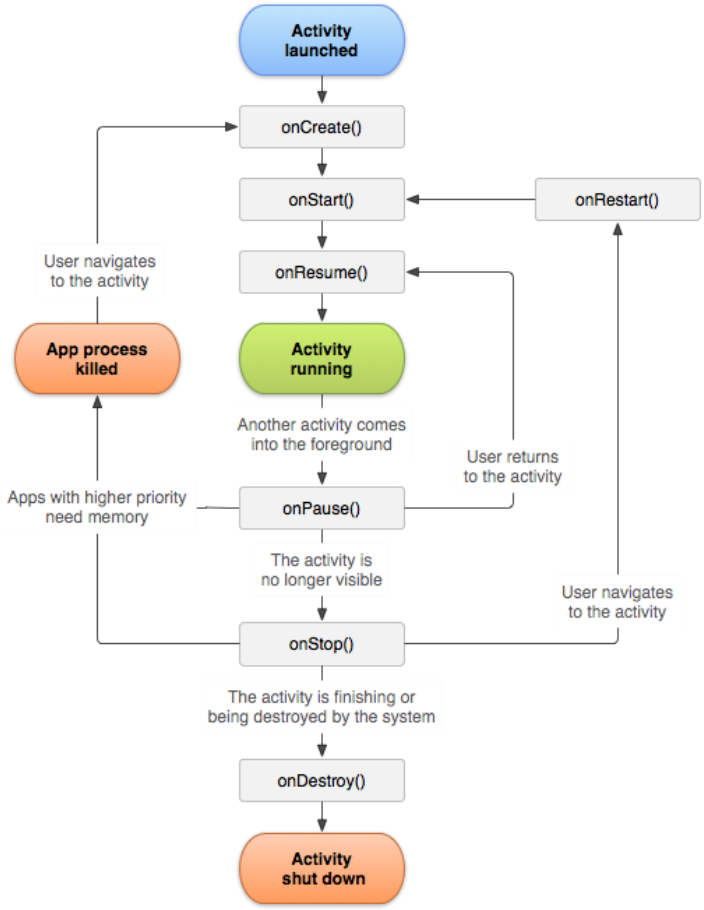

Abbildung 3: Activity-Lebenszyklus [4]

2.4 Entwicklungsumgebung und Tools

Um Android-Anwendungen schnell und komfortabel entwickeln zu können, werden eine Java-IDE[2] sowie das Android Software Development Kit (SDK) und die Android Development Tools (ADT) benötigt.

Auf Googles Android-Developer-Website [4] steht für diesen Zweck ein Komplettpaket für Entwickler zum Download bereit. Dieses enthält alle Tools, die für die App-Entwicklung notwendig sind:

- Eclipse mit ADT-Plugin
- Android SDK-Tools
- Android Plattform-Tools
- Die neueste Android-Plattform
- Das neueste Android-System-Image für den Geräte-Emulator

Nach dem Entpacken müssen nur noch die gewünschten Entwicklungskomponenten (APIs) installiert werden.

Mithilfe dieses Pakets werden die typischen Aufgaben eines Android-Entwicklers vereinfacht. So ist es mit wenigen Klicks möglich, Projektstrukturen anzulegen, grafische Benutzeroberflächen zu erzeugen oder Debugging zu betreiben.

Auch bei der Problematik der vielen unterschiedlichen Geräte erweist sich das Paket als sehr hilfreich, da es einen stabilen und vielseitigen Emulator bereitstellt, mit dem der Entwickler seine Anwendung auf Geräten mit unterschiedlichen Spezifikationen (Bildschirmgröße, Prozessor etc.) testen kann.

[2] Integrated Development Environment

2.5 Aufbau einer Android-App

An dieser Stelle soll der Aufbau einer Android-App dargestellt und die grundlegenden Prinzipien bei der Erstellung einer solchen vermittelt werden. Folgende Punkte sollen hierbei besonders hervorgehoben werden:

- APIs
- Ressourcen (Layout, Strings, Drawables etc.)
- AndroidManifest
- Activity-Klassen
- Fragment-Klassen

Auf die allgemeine Struktur eines Android-App-Projektes wird an dieser Stelle nicht näher eingegangen, da diese von der IDE+ADT automatisch generiert wird.

2.5.1 Android APIs

Android APIs[3] sind von Google angebotene Programmierschnittstellen. Zu jeder neuen Version des Betriebssystems werden neue APIs bereitgestellt, auf die Entwickler zugreifen können. Auf der Google-Developer-Website [4] sind die neuesten Änderungen aufgelistet.

Konkret stellen die APIs eine Sammlung von vom System bereitgestellten Klassen dar, welche den Zugriff auf bestimmte Funktionen ermöglichen. Es ist jedoch zu beachten, dass viele der neuen Funktionen nicht auf Geräten mit älteren Android-Versionen zur Verfügung stehen. Aus diesem Grund kann beim Erstellen des Projekts das „Minimum Required SKD“ angegeben werden, was die älteste Android-Version angibt, unter der die Anwendung lauffähig sein soll. Werden beim Programmieren Funktionen benutzt, welche

[3] Application Programming Interface

„In der R.java-Datei werden alle Ressourcen mit ihren IDs verwaltet. “

nicht von diesem SDK unterstützt werden, so weist die IDE+ADT den Entwickler darauf hin.

2.5.2 Ressourcen

In Android-Anwendungen können unterschiedliche Ressourcen eingebunden werden. Dazu gehören zum Beispiel Bilder, Audiodateien oder Textfelder. Dadurch kann eine Anwendung aufwändiger gestaltet und internationalisiert werden, indem für verschiedene Sprachen Textressourcen für die GUI hinterlegt werden.

ADT kümmert sich um die automatische Verwaltung von Ressourcen. Es legt eine festgelegte Ordnerstruktur im Projekt an und erstellt die *R.java*-Datei, welche allen Ressourcen feste IDs zuordnet. Die Ordnerstruktur kann vom Entwickler um Unterordner erweitert werden, die *R.java*-Datei sollte jedoch nicht verändert werden – nur so ist sichergestellt, dass auf alle Ressourcen problemlos zugegriffen werden kann.

Mit der steigenden Anzahl von unterschiedlichen Geräten wuchs auch die Ordnerstruktur der App-Ressourcen. Mussten anfänglich noch Geräte mit sehr ähnlichen Spezifikationen bedient werden, so gibt es heute vor allem bei den Bild-Ressourcen viele unterschiedliche Dateien in verschiedenen Größen und Auflösungen für eine optimierte Anzeige auf verschiedenen Geräten. Auch durch die Unterstützung von Tablets seit Android 3.x wurde der Verzeichnisbaum erweitert und besitzt nun *layout*-Ordner mit dem Zusatz „large“ für die Bildschirmgröße.

Abbildung 4 zeigt die von ADT erzeugte Ordnerstruktur für Ressourcen.

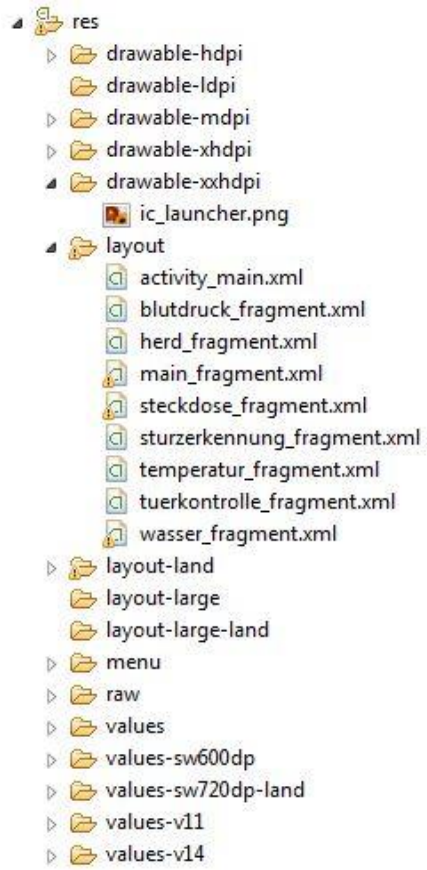

Abbildung 4: Ordnerstruktur für Ressourcen

2.5.3.1 XML-Dateien

Bei der Android-Programmierung spielen XML-Dateien eine wichtige Rolle. Die Auszeichnungssprache wird benutzt, um grafische Oberflächen, Farbtabellen, Wertetabellen (z.B. für Strings, Integers) und Ähnliches zu definieren. Die Dateien werden in der App-Ordnerstruktur an den entsprechenden Stellen abgelegt und können direkt in der IDE bearbeitet werden. Die wichtigste XML-Datei einer Android-Anwendung wird im nächsten Abschnitt behandelt.

2.5.3.2 AndroidManifest.xml

Die *AndroidManifest.xml* liefert dem Betriebssystem wichtige Informationen zu der Anwendung, ohne welche die App nicht ausgeführt werden kann. Dazu zählen zum Beispiel der Name und die Berechtigungen der Anwendung sowie die minimale respektive die Zielversion des Android-SDK.

Listing 1 zeigt beispielhaft eine *AndroidManifest*-Datei für eine App mit der Berechtigung für den Internetzugriff.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.testapp"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8"
              Android:targetSdkVersion=”16) />
    <uses-permission android:name=”android.permission.INTERNET”/>
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".Main"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category an-
droid:name="android.intent.category.LAUNCHER" />
            </intent-filter> </activity> </application>
</manifest>
```

Listing 1: AndroidManifest.xml

2.5.4 Activity-Klassen

Activity-Klassen sind für die Darstellung der Benutzeroberfläche zuständig. In früheren Android-Versionen war in der Tat jede Seite, die der Benutzer sah, eine eigene Activity. Seit Android 3.x und der damit eingeführten Tablet-Unterstützung werden Benutzeroberflächen meistens mit Fragments aufgebaut (siehe nächsten Abschnitt).

In jeder Android-App gibt es eine MainActivity, welche den Einstiegspunkt in die Anwendung darstellt. Diese lädt das erste Fragment, beinhaltet die Callback-Methoden für den Zustandswechsel und kann Methoden zur Ereignisverarbeitung enthalten, falls diese mithilfe des *onClick*-Attributs in der XML-Datei des Layouts definiert wurde.

```
public class MainActivity extends FragmentActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

   if(findViewById(R.id.fragment_container)!=null)
   {}
   else

           if(savedInstanceState != null)
           {
                   return;
           }

   MeinFragment meinFrag = new MeinFragment();

   getSupportFragmentManager().beginTransaction().add(R.id.fragment_container,
meinFrag).commit();
    }

   public void nextFrag(View view)
   {
           MeinFragment2 meinFrag2 = new MeinFragment2();

           FragmentTransaction transaction = getSupportFragmentMana-
ger().beginTransaction();

           transaction.replace(R.id.fragment_container, meinFrag2);
           transaction.addToBackStack(null);
           transaction.commit();


   }
}
```

Listing 2: MainActivity mit Fragment-Unterstützung

„Ein Fragment ist eine Art „Unter-Activity“, welche selbst Ereignisse verarbeitet, dynamisch erzeugt und zerstört werden kann und sogar einen eigenen Lebenszyklus besitzt.“

Mit *setContentView()* wird ein leeres Layout in die MainActivity geladen, welches die ID „fragment_container“ besitzt. Danach erfolgt eine Abfrage, ob ein solcher existiert. Ist dies der Fall, so wird mithilfe des *SupportFragmentManagers* (benötigt für die Fragment-Unterstützung bei Android 2.x) und seiner Methode *beginTransaction().add()*dem Layout ein Fragment hinzugefügt.

Abschließend ist die Implementierung einer Methode zur Ereignisverarbeitung zu sehen. Wurde in der XML-Datei des Layouts einem Button das Attribut *onClick* (in diesem Fall android:*onClick=“nextFrag“*) hinzugefügt, so wird in der zugehörigen Activity nach dieser Methode gesucht. In dem dargestellten Programm ersetzt diese Methode ein Fragment durch ein anderes.

2.5.5 Fragment-Klassen

Ein Fragment ist eine Art „Unter-Activity“, welche selbst Ereignisse verarbeitet, dynamisch erzeugt und zerstört werden kann und sogar einen eigenen Lebenszyklus besitzt. Fragments können mit oder ohne grafische Oberfläche erzeugt werden, brauchen jedoch immer als „Wirt“ eine Activity, deren Lebenszyklus sie unterstellt sind. Der Einsatz von Fragments statt Activities hat einerseits den Nachteil, dass sie komplizierter zu implementieren sind, bietet jedoch andererseits auch viele Vorteile. Zum Beispiel können in einer Activity mehrere Fragments laufen, was das Erstellen komplexer Layouts vereinfacht. Auch die Navigation innerhalb der Anwendung wird vereinfacht, da mit der Methode *transaction.addToBackStack()* der aktuelle Zustand eines Fragments gespeichert werden kann, sodass der Nutzer jederzeit mithilfe des „Zurück“-Buttons dorthin zurückkommt. Nicht zuletzt bieten Fragments einen Geschwindigkeitsvorteil, da nicht ständig ganze Activities erzeugt und wieder zerstört werden müssen.

Listing 3 zeigt die Implementierung einer Fragment-Klasse, die ein sichtbares Layout lädt.

```
public class Main_Fragment extends Fragment
{
       @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup contai-
ner,
         Bundle savedInstanceState) {
              //**Layout laden**//
              return inflater.inflate(R.layout.main_fragment, container,
false);
    }
}
```

Listing 3: Fragment-Klasse

3 Ausgangssituation

Wie in der Einleitung erwähnt, baut das vorliegende Praxisprojekt auf den Projekten und Thesen von Thomas Schmitz und Niels Wientzek auf. In diesem Kapitel soll nun die Ausgangssituation beleuchtet werden.

Als Grundlage für das Gesamtsystem stand der Quellcode aus Niels Wientzeks Bachelor-Thesis zur Verfügung. Dieser umfasste die folgenden Punkte:

- Felix-OSGi-Framework[4] für Android
- Felix-Manager-App
- Wassermelder-OSGi-Bundle
- Wassermelder-App

Einzelheiten zu der Implementierung können in der Literatur unter [2] und [3] nachgelesen werden. Die grundsätzliche Funktionsweise des gegebenen Systems lässt sich wie folgt beschreiben:

Mithilfe der Felix-Manager-App wird ein OSGi-Framework auf dem Android-Gerät gestartet, wobei auch ein automatischer Start beim Hochfahren des Gerätes ausgeführt wird. Dabei werden mehrere für den Betrieb des Frameworks benötigte Komponenten als OSGi-Bundles[5] geladen. Darunter befindet sich auch das von Niels Wientzek geschriebene *wassersimbundle*, welches die Funktionalität des Wassermelders implementiert.

Ist das Framework gestartet, so läuft es ressourcenschonend im Hintergrund und wartet auf eine Meldung vom Wassersensor. Beim Eintreffen einer solchen, wird vom *wassersimbundle* ein Android-Intent[6] als Broadcast mit dem Betreff „WassermelderBroadcast“ versendet. Dieser enthält die Kennung, ob die Nachricht verschlüsselt ist, die Länge der Nachricht, den Namen des Sensors, die Device-ID und den Status des Geräts („alive“ oder „alarm“).

---

[4] Freies OSGi-Framework der Apache-Community [5]
[5] Eine OSGi-Anwendung kann in mehrere Module (Bundles) aufgeteilt werden
[6] Android-interner Kommunikationsmechanismus. Beim Intent-Broadcast wird ein „Betreff“ vergeben, sodass nur bestimmte Broadcast-Receiver darauf reagieren.

In der zugehörigen Wassermelder-App wurde ein *BroadcastReceiver* definiert, welcher auf den oben genannten Betreff reagiert. Läuft die App im Vordergrund, so wird die dem Intent angehängte Information stets sowohl in Textform, als auch visuell mithilfe eines Bildes auf dem Bildschirm ausgegeben. Ist der Status des Wassermelders „alarm“, so wird die App automatisch gestartet und es werden zusätzlich ein Alarmton und Vibrationsfeedback ausgegeben.

3.1 Anpassung des gegebenen Systems

Im Laufe der Zeit ergaben sich einige Änderungen, nach denen das vorliegende System nicht mehr funktionstüchtig war. Der Aufbau der von den AAL-Geräten versendeten Nachrichten wurde verändert und die meisten Android-Smartphones und Tablets laufen mittlerweile unter der Betriebssystem-Version 4.x, welche teilweise andere Entwurf-Prinzipien für Anwendungen mit sich brachte. Im ersten Schritt der Entwicklung einer Anwendung für die WieDAS-Geräte musste das System an die neuen Gegebenheiten angepasst werden.

3.1.1 Nachrichtenaufbau

Im Anfangsstadium des WieDAS-Projekts gab es für jedes Gerät innerhalb der Musterwohnung lediglich einen Prototypen. Dementsprechend simpel wurden auch die Nachrichten konzipiert, welche die Geräte senden sollten. Folgender Aufbau wurde bei Nachrichten für die Kommunikation mit der Steuerungssoftware verwendet: „[Gerätename]:[Gerätestatus]“. Da es jedoch sehr wahrscheinlich ist, dass in einer Wohnung auch mehrere Geräte einer Art eingesetzt werden (z.B. mehrere Wassersensoren) und die Nachrichten eventuell auch verschlüsselt werden müssen, wurde der Aufbau der Nachrichten verändert und befolgt nun folgendes Schema: „[Verschlüsselungstyp]:[Länge der Nachricht]:[Gerätename]:[Geräte-ID]:[Gerätestatus]“.

Um das System an den neuen Nachrichtenaufbau anzupassen, musste zunächst analysiert werden, wie das zugrundeliegende OSGi-Framework und das verwendete Kommunikationsbundle (siehe Bachelor-Thesis von Thomas Schmitz [6]) die eingehenden UDP-Nachrichten verarbeitet. Dabei wurde festgestellt, dass die Nachricht automatisch in zwei Teile zerlegt wird: die Gerätekennung und eine zusätzliche Nachricht, wobei der erste Doppelpunkt der empfangenen Nachricht als Trennzeichen dient. Da der Quellcode des Kommunikationsbundles nicht vorlag, musste die Lösung auf anderem Wege implementiert werden.

Um die Nachrichten verarbeiten zu können, sollten diese zunächst nach der Kennung bis zum ersten Doppelpunkt gefiltert werden. Stimmt die gesendete mit der erwarteten Kennung überein, so sollte das *wassersimbundle* die Nachricht an die neue WieDAS-App weiterleiten, wo schließlich entschieden wird, welcher Sensor die Nachricht verschickt hat und wo die Daten in der GUI angezeigt werden sollen.

Für die gewünschte Funktionalität mussten lediglich zwei Einträge im *wassersimbundle* angepasst werden. Die frühere Device-ID wird an zwei Stellen innerhalb des Bundles angegeben: in der *ServiceBinder*-Klasse und in der *MANIFEST.MF*-Datei. Ändert man diese von „wm“ zu „p00d“, so werden alle Nachrichten, welche mit dieser Kennung beginnen – in der Musterwohnung ist dies ausschließlich beim Wassermelder der Fall – vom *wassersimbundle* registriert und an die WieDAS-App weitergeleitet. Die Listings [4] und [5] zeigen die Anpassung im Quellcode.

```
public void activate(ComponentContext context) {
       BundleContext bundleContext = context.getBundleContext();
       Bundle bundle = bundleContext.getBundle();
       this.bundleID = bundle.getSymbolicName();
       if (socketEventReg.startServer(wmPort, ISocketEventRegistrati-
on.UDP)) {
           socketEventReg.addSocketEventHandler(this.bundleID, "p00d",
                  "de/fhd/inflab/waterdetector/alarm");
           System.out.println("WM-Plugin: Event Registrated");
       }
    }
```

Listing 4: activate()-Methode der ServiceBinder-Klasse

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Wassersimbundle
Bundle-SymbolicName: de.fhd.inflab.android.wassersimbundle
Bundle-Version: 1.0.0.0
Bundle-Activator: de.fhd.inflab.android.wassersimbundle.Activator
Import-Package: android.app,android.content,android.net,android.os,and
 roid.widget,de.fhd.inflab.android.com.socket.intf,org.osgi.framework;
 version="1.3.0",org.osgi.service.component,org.osgi.service.event
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Service-Component: OSGI-INF/component.xml,OSGI-INF/componentIntf.xml
DeviceID: p00d
```

Listing 5: MANIFEST.MF des *wassersimbundle*

Diesem Prinzip folgend wurden die Bundles für die anderen Geräte entsprechend entworfen und umgesetzt.

3.1.2 Speicherzugriff unter Android 4.x

Damit die Bundles dynamisch zur Laufzeit eingebunden werden können, müssen sie auf dem Gerät im internen respektive externen Speicher abgelegt werden. Dabei treten zwei Probleme auf, deren Lösung in diesem Abschnitt erläutert werden soll:

- Unterschiedliche Verzeichnisstruktur (Hersteller- und Versionsabhängig)
- Ausführung von dex-Dateien[7] außerhalb des App-Addressraums nicht gestattet (neue Android-Versionen)

Es war nicht möglich, das Felix-Framework – welches auf dem Samsung Galaxy S Plus (Android 2.3.3) vollständig lauffähig ist – ohne Anpassungen auf einem Samsung Galaxy S III (Android 4.1.2) laufen zu lassen. Zum einen unterscheidet sich die Verzeichnisstruktur zwischen den beiden Geräten, zum anderen verhindert die neuere Android-Version die Ausführung der dex-Dateien der Bundles, wenn diese nicht im Adressraum der App, sondern auf der SD-Karte respektive dem internen Telefonspeicher liegen.

[7] Ausführbarer Dalvik-Bytecode

Die Problematik der unterschiedlichen Verzeichnisstrukturen lässt sich mithilfe einer Methode aus den Android-APIs umgehen.

```
File sdcard = Environment.getExternalStorageDirectory();
String path = sdcard.getAbsolutePath();
```

Listing 6: Auszug aus der FelixService-Klasse des Felix-Managers

Obwohl der Name der Methode etwas anderes vermuten lässt, liefert *Environment.getExternalStorageDirectory()* den Pfad zum internen Speicher des Smartphones. Dieser kann anschließend in eine String-Variable geschrieben und im Programm statt einer absoluten Pfadangabe verwendet werden.

Seit Android 4 ist es Apps aus Sicherheitsgründen nicht mehr erlaubt, dex-Code vom internen Speicher oder der SD-Karte zu laden beziehungsweise auszuführen. Ein Einbinden der OSGi-Bundles kann also nur erfolgen, wenn diese im Adressraum der App liegen. Um dies zu erreichen, ohne die Bundles zu einem festen Bestandteil der App zu machen (und somit gegen das Grundprinzip von OSGi zu verstoßen), müssen die Bundles zur Laufzeit in ein von dem Felix-Manager erzeugtes Cache-Verzeichnis kopiert und von dort aus gestartet werden. Listing 7 zeigt beispielhaft die Implementierung dieses Vorgangs für ein OSGi-Bundle. Alle Bundles, die im Felix-Framework laufen sollen, müssen auf diese Weise installiert werden.

Nach diesen Modifikationen ist das Framework auf unterschiedlichen Geräten lauffähig und unterstützt die Design-Richtlinien von Android 4[8].

[8] Die aktuellste Version ist Android 4.2 (Stand: April 2013)

```
//**Erzeugen des Cache-Verzeichnisses**//

Properties configProbs = new Properties();

File dexOutputDir = this.getApplicationContext().getDir("dexDir", 0);

configProbs.setProperty("org.osgi.framework.storage", dexOutput-
Dir.getAbsolutePath());

                                        ...

//**Pfade holen**//

File sdcard = Environment.getExternalStorageDirectory();
String path = sdcard.getAbsolutePath();
dtPath = dexOutputDir.getAbsolutePath();

//**Bundle in das Cache-Verzeichnis kopieren**//

try{
     File f1 = new File(path+"/bundles/org.apache.felix.shell-
1.4.2.jar");
     File f2 = new File(dtPath+"/test.jar");
     InputStream in = new FileInputStream(f1);

     OutputStream out = new FileOutputStream(f2);

     byte[] buf = new byte[1024];
     int len;
     while ((len = in.read(buf)) > 0){
             out.write(buf, 0, len);
           }
     in.close();
     out.close();
    }
catch(FileNotFoundException ex){
      System.out.println(ex.getMessage() + " in the specified directo-
ry.");
      System.exit(0);
     }
catch(IOException e){
      System.out.println(e.getMessage());
     }
                                        ...
//**Bundle installieren**//
Bundle shellBundle = fe-
lix.getBundleContext().installBundle("reference:file:/"+dtPath+"/test.ja
```

Listing 7: Auszug aus der FelixService-Klasse des Felix-Managers

4 Konzept der WieDAS-Anwendung

Bevor eine Anwendung entwickelt werden kann, müssen die Anforderungen an die Benutzeroberfläche und die Funktionalität konkretisiert werden. Das Grundkonzept für die WieDAS-App soll Thema dieses Kapitels sein. Abbildung 5 zeigt das Use-Case-Diagramm der Anwendung.

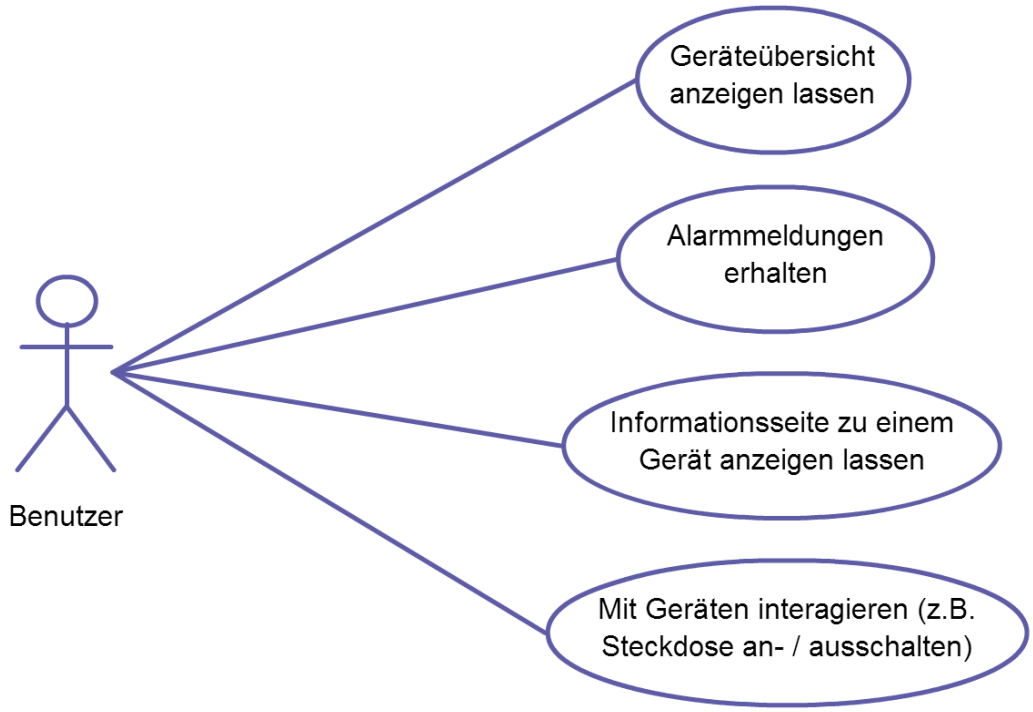

Abbildung 5: Use-Case-Diagramm der WieDAS-App

4.1 Grafische Benutzeroberfläche

Die zu entwickelnde Anwendung soll den Zugriff auf alle Geräte und Sensoren der WieDAS-Musterwohnung über eine ansprechende und funktionale Benutzeroberfläche ermöglichen. Denkbar sind dabei unterschiedliche Ansichten für den Einsatz auf einem Smartphone respektive Tablet. Dieses Praxisprojekt soll sich jedoch auf eine Smartphone-GUI (im Hoch- und Querformat) beschränken.

„Die grafische Oberfläche wurde in zwei Komponenten aufgeteilt - eine Startseite, die alle Geräte auflistet und jeweils eine Informationsseite pro Gerät.“

Die durchschnittliche Bildschirmgröße bei aktuellen Smartphones beträgt in etwa 4,6 Zoll in der Diagonalen. Bei dieser Größe wäre die Anzeige aller Geräte auf einer Seite nicht akzeptabel – vor allem ältere Menschen (an die das System gerichtet ist) würden den Überblick verlieren und so eine Anwendung nicht nutzen wollen. Aus diesem Grund soll die grafische Oberfläche der Anwendung in zwei Komponenten aufgeteilt werden – eine Startseite, die alle Geräte auflistet und jeweils eine Informationsseite, die das Gerät identifiziert, den Status anzeigt, Alarmmeldungen ausgibt und sonstige Funktionen dem Nutzer zur Verfügung stellt.

Die konkrete Implementierung einer solchen Benutzeroberfläche wird im Kapitel „Realisierung“ vorgestellt.

4.2 Funktionalität

Im Rahmen des WieDAS-Projekts wurden vom Labor für Informatik an der FH-Düsseldorf insgesamt sieben Sensoren / Geräte für den Einsatz im Bereich des Ambient Assisted Living entwickelt:

- Wassermelder
- Steckdosenkontrolle
- Temperatursensor
- Türsteuerung / -überwachung
- Herdüberwachung
- Blutdruckmessgerät
- Sturzsensor

Jede Informationsseite (außer die Türüberwachung) muss den Namen des Geräts, seine ID und die Adresse anzeigen. Nachfolgend soll die Funktionalität der Informationsseite zu den jeweiligen Geräten genauer skizziert werden.

Wassermelder

Der Status des Sensors soll in kurzen Intervallen aktualisiert und angezeigt werden. Sendet der Wassersensor eine Alarmmeldung, so soll das Smartphone neben einer visuellen auch eine akustische und haptische Warnung ausgeben. Ist die App zu diesem Zeitpunkt nicht aktiv, soll sie automatisch gestartet (dieser Teil ist in der Bachelor-Thesis von Niels Wientzek [3] beschrieben) und die Informationsseite des Wassermelders angezeigt werden.

Steckdosenkontrolle

Die Informationsseite soll stets den aktuellen Status der Steckdose anzeigen. Zudem soll der Benutzer diese von hier aus an- und ausschalten können.

Temperatursensor

Diese Seite soll lediglich über die aktuelle Temperatur informieren und diese anzeigen.

Türsteuerung / -überwachung

Der Benutzer soll die Möglichkeit haben, jederzeit auf den Videostream der Überwachungskamera zugreifen zu können. Als zusätzliche Funktion soll es möglich sein, aus der App heraus die Tür zu öffnen und zu schließen.

Herdüberwachung / Blutdruckmessgerät / Sturzsensor

Die Funktion dieser Sensoren ist der von dem Wassermelder sehr ähnlich. Die Informationsseiten können analog dazu aufgebaut werden.

5 Realisierung

Nachdem die Anforderungen an die Anwendung definiert wurden, soll in diesem Kapitel die konkrete Realisierung der Anwendung beschrieben werden. Dabei soll zunächst auf die Erstellung der grafischen Oberfläche und anschließend auf die Implementierung der einzelnen OSGi-Bundles eingegangen werden.

5.1 GUI

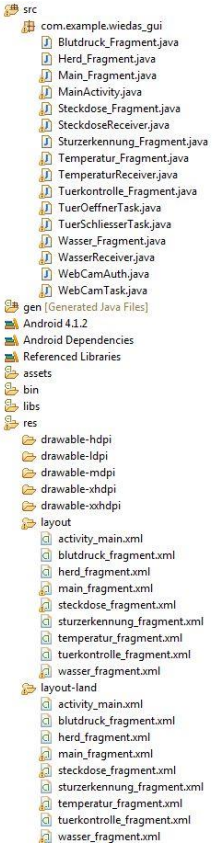

Abbildung 6: Projektstruktur

Die grafische Benutzeroberfläche wurde vollständig durch Fragments realisiert. Dabei besitzt jedes Fragment ein eigenes Layout (Hoch- und Querformat) sowie im Falle der Informationsseiten zusätzlich einen eigenen *BroadcastReceiver* für Intents. Die Methoden für die Ereignisverarbeitung sind – wie in Kapitel 2.5.4 beschrieben – in der *MainActivity*-Klasse definiert. Die Projektstruktur wird in der Abbildung 6 gezeigt.

5.1.1 Startseite

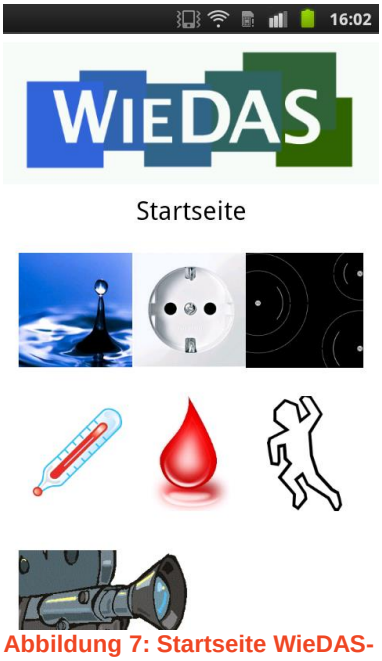

Abbildung 7: Startseite WieDAS-App

Beim Starten der Anwendung, wird die *MainActivity* erzeugt, welche zunächst ein leeres Layout besitzt. Diese lädt anschließend das *Main_Fragment*, das mit dem zugehörigen Layout die Startseite der Anwendung darstellt. Auf dieser wird eine Übersicht der einzelnen Geräte präsentiert. Abbildung 7 zeigt das Layout der Startseite wie es auf einem Samsung Galaxy S Plus angezeigt wird. Durch Antippen der Symbole kommt der Nutzer auf die zugehörige Informationsseite der Geräte. Im Listing 8 ist beispielhaft die Implementierung der *goToWasser()*-Methode zu sehen, welche beim Antippen des Wassermelder-Symbols aufgerufen wird und die Informationsseite des Geräts – respektive das *Wasser_Fragment* – lädt und dabei die Transaktion zum *BackStack* hinzufügt, was eine Navigation zurück zur Startseite mithilfe der Zurück-Taste ermöglicht.

```
public void goToWasser(View view)
   {
          Wasser_Fragment wasserFrag = new Wasser_Fragment();

          android.support.v4.app.FragmentTransaction transaction = get-
SupportFragmentManager().beginTransaction();

          transaction.replace(R.id.fragment_container, wasserFrag, "was-
ser_frag");
          transaction.addToBackStack(null);
          transaction.commit();


   }
```

Listing 8: Methode zum Laden des *Wasser_Fragments*

Eine Besonderheit stellt die Alarmmeldung eines Gerätes dar. Sendet zum Beispiel der Wassersensor eine Alarm-Nachricht, so empfängt der *Wasser-Receiver* einen Intent und startet die WieDAS-Anwendung. Hier muss nun dafür gesorgt werden, dass die App direkt die Informationsseite des Wassermelders lädt. Das Diagramm in Abbildung 8 zeigt den Ablauf des Startvorgangs der Anwendung.

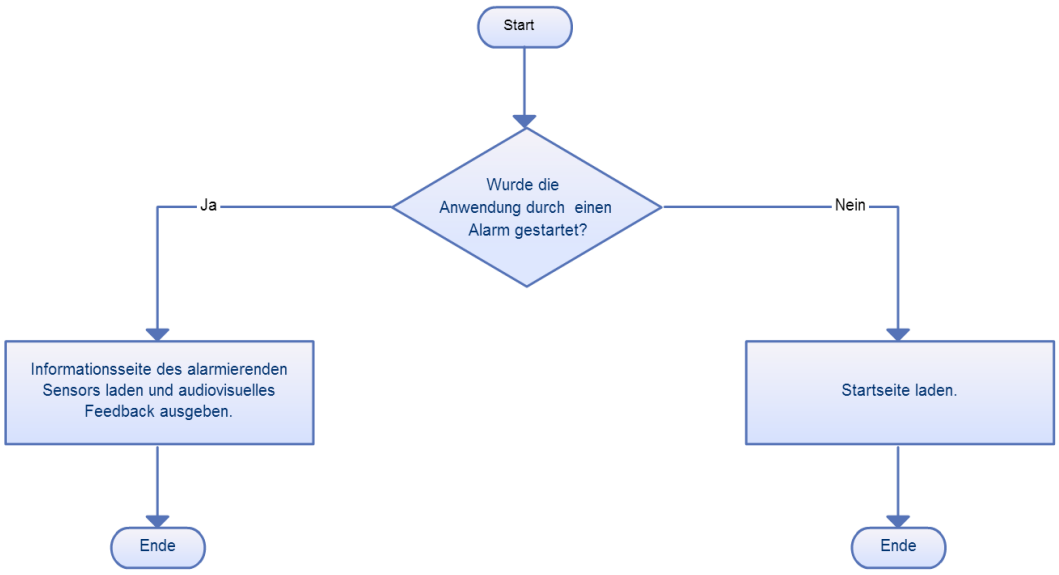

Abbildung 8: Ablaufdiagramm zum Anwendungsstart

Diese Funktionalität wurde mithilfe einer einfachen Int-Variablen und der *onResume()*-Methode der *MainActivity* realisiert. Empfängt der *WasserReceiver* einen „Alarm-Intent“, setzt er eine *lock*-Variable und startet die WieDAS-

App. Deren *MainActivity* entscheidet dann anhand dieser Variablen, welche Aktion durchgeführt werden soll. Im Rahmen dieses Praxisprojekts wurde die Alarm-Funktion für den Wassermelder realisiert. Listing 9 zeigt die konkrete Implementierung.

```
public void onResume()
   {
            super.onResume();

            //**Abfrage, ob das Main_Fragment bereits erzeugt wurde**//
            if(lock == 0)
            {
            //**Main_Fragment erzeugen**//
            Main_Fragment mainFrag = new Main_Fragment();

            getSupportFragmentManager().beginTransaction().add(R.id.fragment_container, main-
Frag, "main_frag").commit();
            lock = 1;
            }

            //**Abfrage, ob ein Alarm ausgelöst wurde - die lock-Variable wird im WasserRe-
ceiver gesetzt**//
            //**Bei Alarm wird das Wasser_Fragment direkt geladen**//
            if(lock == 2)
            {
                Wasser_Fragment wasserFrag = new Wasser_Fragment();

                    android.support.v4.app.FragmentTransaction transaction = getSupportFrag-
mentManager().beginTransaction();

                    transaction.replace(R.id.fragment_container, wasserFrag, "wasser_frag");
                    transaction.addToBackStack(null);
                    transaction.commit();

                    //**Audio-Feedback bei Alarm**//
                    final AudioManager audioManager = (AudioManager)
this.getSystemService(this.AUDIO_SERVICE);
           int MusicMaxVol = audioManager.getStreamMaxVolume(audioManager.STREAM_MUSIC);
           final int MusicVol = audioManager.getStreamVolume(audioManager.STREAM_MUSIC);
           //Max Lautstärke größer als aktuelle?
           if(MusicMaxVol > MusicVol){
                 //wenn ja, aktuelle auf Max stellen
                 audioManager.setStreamVolume(audioManager.STREAM_MUSIC, MusicMaxVol, audio-
Manager.STREAM_MUSIC);
                 }
           MediaPlayer mp = MediaPlayer.create(this, R.raw.alarm);
           mp.start();
        mp.setOnCompletionListener(new OnCompletionListener() {

                  @Override
                  public void onCompletion(MediaPlayer mp) {
                         //Nach dem abspielen Instanz freigeben und Lautstärke zurücksetzen
                     mp.release();
                     audioManager.setStreamVolume(audioManager.STREAM_MUSIC, MusicVol,
audioManager.STREAM_MUSIC);
                  }
               });

                    lock = 1;
            }
   }
```

Listing 9: onResume()-Methode der MainActivity

5.1.2 Informationsseite

Die Informationsseiten der einzelnen Geräte sind prinzipiell ähnlich aufgebaut, bieten jedoch jeweils eine auf das Gerät angepasste Oberfläche. Die zugehörigen Fragments implementieren Methoden um die empfangenen Daten zu verarbeiten und ihre Oberfläche zu aktualisieren. Die Basis dafür liefert der Quellcode aus Niels Wientzeks Bachelor-Thesis [3], welcher auf die Fragment-Architektur und für die unterschiedlichen Geräte angepasst wurde. Ein Fragment verhält sich hierbei ähnlich wie eine Activity, es ist jedoch darauf zu achten, dass die richtigen Methoden für die Steuerung des Lebenszyklus (Kapitel 2.3) benutzt werden und dass der Zugriff auf das Layout über die Methode *getView()* erfolgen muss. Die Implementierung einer Informationsseite als Fragment soll an dieser Stelle am Beispiel des Wassermelders gezeigt werden.

Abbildung 9: Informationsseite Wassermelder

Jede Informationsseite (abgesehen von der Türkontrolle) besitzt eine Methode zum Setzen der Geräteinformationen, welche von dem zugehörigen Receiver aufgerufen wird. Aufgrund der in Kapitel 3.1.1 aufgezeigten Änderungen können die einzelnen Strings *deviceID*, *address* und *data* nicht einfach übernommen werden. Da die Hauptinformationen in dem *data*-String stehen, muss dieser von der *setWasser*()-Methode in seine Teilstrings aufgespalten und neu abgespeichert werden. Dieser Vorgang ist in Listing 10 zu sehen.

Zuerst erfolgt die Abfrage, ob die lokale Variable *dataA* bereits Daten enthält. Wurde sie vom *WasserReceiver* beschrieben, so enthält sie beim ersten Wassersensor den String „wm:0001:[Status]“.

Mithilfe der *indexOf()*-Methode wird nun die Position des nächsten Doppelpunkts ermittelt und der gewünschte Teilstring unter Zuhilfenahme der *sub-*

string()-Methode in eine neue Variable geschrieben. Die neuen Strings werden dann in der *updateUI()*-Methode weiterverarbeitet und für die Anzeige der Gerätedaten benutzt.

```
public void setWasser(String deviceID, String address, String data) {

          deciveIDA  = deviceID;
          addressA = address;
          dataA = data;

          if(dataA.indexOf(":") != -1)
          {
          endpos = dataA.indexOf(":", 0);
          deciveIDA = dataA.substring(0, endpos);
          startpos = endpos+1;
          endpos = dataA.indexOf(":", startpos);
          deviceNr = dataA.substring(startpos, endpos);
          startpos = endpos+1;
          dataA = dataA.substring(startpos);
          }
     }
```

Listing 10: setWasser()-Methode des Wasser_Fragments

Die *updateUI()*-Methode enthält eine Abfrage, ob sich tatsächlich der richtige Sensor gemeldet hat (siehe Problematik in Kapitel 3.1.1) und aktualisiert daraufhin entsprechend die grafische Oberfläche.

Die *getView().findViewById()*-Methode liefert über die in der *wasser_fragment.xml* zugewiesenen IDs die TextView und die ImageView des Layouts, welche dann lokalen Variablen zugewiesen werden. Hat die vorangegangene Abfrage ergeben, dass die Daten vom Wassersensor kommen, werden diese in die TextView geschrieben. Danach folgt eine Abfrage, welchen Status der Sensor hat und ein entsprechendes Bild (rot, gelb oder grün) wird angezeigt. Falls keine Daten vorliegen, werden die Textfelder mit Nullen gefüllt und die gelbe Grafik geladen.

Es können auch mehrere Wassersensoren angezeigt werden. Dabei hilft eine einfache *case*-Anweisung, welche anhand der Gerätenummer respektive Geräte-ID unterschiedliche TextViews und ImageViews lädt.

Analog zu dieser Vorgehensweise können die Informationsseiten der anderen Sensoren realisiert werden.

```
public void updateUI(){
          TextView tv = (TextView) get-
View().findViewById(R.id.steckdoseTxt);
          if(deciveIDA.equals("wm"))
          tv.setText("Gerät: "+deciveIDA+"\n"+"Gerätenummer:
"+deviceNr+"\n"+"Adresse: "+addressA+"\n"+"Status: "+dataA);
          else
          tv.setText("Info 0 0 0");
          ImageView status = (ImageView) get-
View().findViewById(R.id.steckStatus);
          if (dataA.equals("alive")) sta-
tus.setImageResource(R.drawable.status_gruen);
          if (dataA.equals("alarm")) sta-
tus.setImageResource(R.drawable.status_rot);
   }
```

Listing 11: updateUI()-Methode des Wasser_Fragments

5.1.3 GUI aktualisieren

Das automatische Aktualisieren der GUI erfolgt analog zum von Niels Wientzek verwendeten Verfahren mittels einer *Runnable*-Instanz und eines Handlers.

Die *run()*-Methode der *Runnable*-Instanz ruft *updateUI()* auf und erzeugt anschließend unter Zuhilfenahme eines Handlers (hier *m_handler*) und seiner *postDelayed*-Methode einen Eintrag in der Message Queue von Android. Dies sorgt dafür, dass die *run()*-Methode der angegebenen Instanz nach einer definierten Zeit erneut aufgerufen wird. Listing 12 zeigt die Implementierung der Lösung.

```
Runnable m_statusChecker = new Runnable(){
        @Override
        public void run() {
             updateUI();
             m_handler.postDelayed(m_statusChecker, m_interval);
        }
        };
```

Listing 12: Automatisch wiederholter Aufruf der updateUI()-Methode

Damit die Endlosschleife wieder beendet wird, muss der Eintrag aus der Message Queue wieder entfernt werden. Dies kann durch einen Aufruf der Methode *removeCallbacks()* erreicht werden.

5.2 Intent-Receiver

Mit Ausnahme der Türkontrolle (diese wird im Kapitel „Ausnahmefall Türüberwachung“ behandelt) besitzt jedes Gerät innerhalb der WieDAS-App einen *BroadcastReceiver*, der auf von den OSGi-Bundles versandte Intents reagiert. Der Aufbau eines solchen soll in diesem Abschnitt am Beispiel des *WasserReceivers* erklärt werden.

Damit ein Intent-Receiver funktioniert, muss dieser dem Android-Betriebssystem bekannt sein. Hierzu ist eine Registrierung notwendig. Am einfachsten geht dies über einen Eintrag in der *AndroidManifest.xml*-Datei. Mit den in Listing 13 gezeigten Zeilen wird der *WasserReceiver* registriert und bekommt als *Intent-Filter* den Betreff „WassermelderBroadcast“.

```
<receiver android:name=".WasserReceiver">
            <intent-filter>
                  <action android:name="WassermelderBroadcast" />
                  <category andro-
id:name="android.intent.category.DEFAULT" />
            </intent-filter>
</receiver>
```

Listing 13: Registrierung des WasserReceivers in der AndroidManifest.xml-Datei

Wird nun ein Intent mit diesem Betreff per Broadcast an das System geschickt, so wird automatisch die *onReceive()*-Methode der *WasserReceiver*-Klasse ausgeführt.

Zu Beginn dieser Methode werden die dem Intent angehängten Gerätedaten ausgelesen und abgespeichert. Anschließend werden sie mithilfe der *setWasser()*-Methode dem *Wasser_Fragment* übergeben.

```
Bundle extras = intent.getExtras();
//Daten aus dem Intent in Strings schreiben
String deviceID = extras.getString("deviceID");
String address = extras.getString("address");
String data = extras.getString("data");
//Instanz des Fragments erzeugen und Daten übergeben
Wasser_Fragment wa = new Wasser_Fragment();
wa.setWasser(deviceID, address, data);
```

Listing 14: Übergabe des Intent-Anhangs an das Wasser_Fragment

Mit einer solchen Implementierung ist es nun möglich, jeder Informationsseite die Gerätedaten zu übermitteln. Nachdem die angehängten Daten wieder in Teilstrings zerlegt wurden, ist es mithilfe einer if-Abfrage möglich, eine Alarmmeldung auszugeben. Der akustische Alarm wurde bereits im Kapitel 5.1.1 beschrieben. Das haptische Feedback lässt sich durch Vibration realisieren und wird in Listing 15 beschrieben.

```
Vibrator v = (Vibra-
tor)context.getSystemService(context.VIBRATOR_SERVICE);
// Vibrations Muster als Arry defenieren
long vibrateCode[] = {200, 200, 200, 200, 200, 200, 200};
// Anfangen nach Muster zu vibrieren, ohne Wiederholung
v.vibrate(vibrateCode, -1);
```

Listing 15: Haptisches Feedback durch Vibration

Im Falle des Wassermelders soll es jedoch zusätzlich möglich sein, bei einem Alarm die WieDAS-App automatisch zu starten und die Informationsseite des Wassermelders zu laden.

Wie in [3] beschrieben, lässt sich die *MainActivity* mithilfe eines *appStartIntent* automatisch aufrufen. Damit bei der neuen WieDAS-Anwendung direkt die Wassermelder-Informationsseite geladen wird, muss eine Lock-Variable gesetzt werden – der restliche Ablauf ist in Kapitel 5.1.1 beschrieben. Der benötigte Intent wird mithilfe des *PacketManagers* generiert und verschickt. Die Implementierung ist in Listing 16 zu sehen.

```
MainActivity main = new MainActivity();
main.setLock(2);

                                    ...

PackageManager pm = context.getPackageManager();
Intent appStartIntent =
pm.getLaunchIntentForPackage("com.example.wiedas_gui");
     if (null != appStartIntent){
         context.startActivity(appStartIntent);
        }
```

Listing 16: Automatischer Aufruf der MainActivity mit direktem Laden des Wasser_Fragments

Die gezeigte Vorgehensweise kann bei Bedarf auf weitere Geräte angewendet werden, sodass auch zum Beispiel der Sturzsensor einen Alarm auslösen und die Anwendung automatisch starten kann.

5.3 OSGi-Bundles

Ein Bundle ist ein Java-Archiv mit Zusatzinformationen in einer Manifest-Datei. Bei dem WieDAS-Projekt werden gerätespezifische Bundles benutzt, um die Sensordaten über ein OSGi-Event vom Felix-Framework zu bekommen und als Intent an die WieDAS-App zu senden.

Die *Activator*-Klasse implementiert das Interface der *BundleActivator*-Klasse und besitzt die Methoden *start()* und *stop()*, welche beim Starten und Stoppen des Bundles ausgeführt werden, sofern die Klasse in der Manifest-Datei angegeben wurde.

OSGi-Bundles wurden in Niels Wientzeks Bachelor-Thesis [3] detailliert beschrieben, weswegen in diesem Kapitel lediglich auf den anwendungsbezogenen Einsatz eingegangen werden soll.

5.3.1 Erstellen von Android-kompatiblen Bundles

Da OSGi-Bundles Java-kompilierte Programme sind, müssen sie für den Einsatz unter Android in dex-Bytecode umgewandelt werden. Hierzu wird das Bundle zunächst als .jar-Archiv in der IDE exportiert. Danach muss unter Zuhilfenahme des SDK-Tools *dx* der Dalvik-Bytecode erzeugt und anschließend mithilfe von *aapt* wieder dem Archiv hinzugefügt werden. Die dazu nötigen Befehle werden in der Konsole eingegeben und sind im Listing 17 aufgeführt.

```
java –jar dx.jar –dex –output=classes.dex C:\bunlde.jar

aapt add C:\bundle.jar classes.dex
```

Listing 17: Erzeugen eines Android-kompatiblen Bundles

5.3.2 Bidirektionale Kommunikation mithilfe von OSGi-Bundles

Je nach Anwendungsfall kann es nötig sein, nicht nur Daten von einem Gerät zu empfangen, sondern auch Daten / Befehle an das Gerät zu senden. Am Beispiel der Steckdosenkontrolle soll die bidirektionale Kommunikation zwischen einem Gerät und der WieDAS-App erklärt werden. Abbildung 10 zeigt das Klassendiagramm des *Steckdosebundles*, welches vom *wassersimbundle* aus der Arbeit von Niels Wientzek [3] abgeleitet wurde.

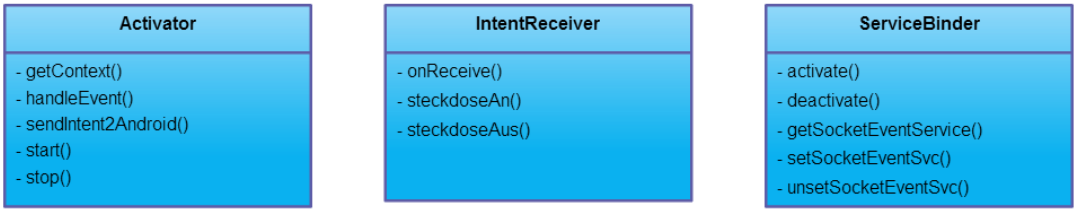

Abbildung 10: Klassendiagramm des Steckdosebundles

5.3.2.1 Empfangen der Gerätedaten

Die *Activator*-Klasse des Bundles implementiert die Methoden *handleEvent()* und *sendIntent2Android()*. Mit diesen ist es möglich, die empfangenen Gerätedaten, welche zu Beginn lediglich als OSGi-Event im Framework zur Verfügung stehen, in Strings abzuspeichern und diese als Android-Intent an die WieDAS-App zu senden. Dieser Vorgang ist für alle Geräte gleich, weswegen jedes Gerätebundle diese Methoden in der gleichen Weise implementiert. Listing 18 und 19 zeigen die Lösung im Detail.

```
public void handleEvent(Event event){

	System.out.println("Steckdose OSGi Event empfangen");
	//OSGi Event Daten in Lokale Strings kopieren
	String deviceID = (String) event.getProperty("id");
	String address = (String) event.getProperty("address");
	String data = (String) event.getProperty("data");
	//Intent mit den Daten senden
	sendIntent2Android(deviceID,address,data);

	}
```

Listing 18: handleEvent()-Methode des Steckdosebundles

```
public static void sendIntent2Android(String deviceID, String address,
String data){

	Intent intent = new Intent();
	intent.setAction("SteckdoseBroadcast");
	//Übergabe Strings ans Intent hängen
	intent.putExtra("deviceID", deviceID);
	intent.putExtra("address", address);
	intent.putExtra("data", data);
	//Intent senden
	androidContext.sendBroadcast(intent);
	}
```

Listing 19: sendIntent2Android()-Methode des Steckdosebundles

5.3.2.2 Daten an ein Gerät senden

Die Kommunikation zwischen der WieDAS-App und dem Gerät wird mithilfe eines Intents angestoßen. Die Buttons der Informationsseite nutzen zur Ereignisverarbeitung die Methoden *steckdoseAn()* und *steckdoseAus()*, welche einen Intent mit dem Betreff „SteckdoseIntentBroadcast“ mit dem String „an“ respektive „aus“ versehen und an das System rausschicken.

```
public void steckdoseAn(View view)
   {
	Intent intent = new Intent();
		intent.setAction("SteckdoseIntentBroadcast");
		intent.putExtra("steckDaten", "an");
		//Übergabe Strings an Intent hängen
		//Intent senden
		con.sendBroadcast(intent);
   }
```

Listing 20: steckdoseAn()-Methode der MainActivity

Dieser Intent muss nun von dem Steckdosebundle erfasst und verarbeitet werden. Da das Bundle jedoch keine eigenständige Android-App ist, lässt sich der IntentReceiver nicht wie in Kapitel 5.2 in der *AndroidManifest*-Datei registrieren. Stattdessen muss die Registrierung zur Laufzeit erfolgen. Dies kann innerhalb der *start()*-Methode der Activator-Klasse erfolgen. Listing 21 zeigt die Registrierung eines IntentReceivers zur Laufzeit.

```
IntentFilter osgiSteckdoseSchalterFilter = new IntentFilter();

osgiSteckdoseSchalterFilter.addAction("SteckdoseIntentBroadcast");

androidContext.registerReceiver(IntentReceiver, osgiSteckdoseSchalter-
Filter);
```

Listing 21: Registrierung eines IntentReceivers zur Laufzeit

Zuerst wird eine neue Instanz des IntentFilters erzeugt. Danach wird über die Methode *addAction()* festgelegt, auf welchen Betreff der Receiver reagieren soll. Im letzten Schritt wird der Receiver samt zugehörigem Betreff beim Android-System registriert und kann – wenn das Bundle gestoppt wird – mittels der Methode *unregisterReceiver()* wieder abgemeldet werden.

Die Klasse *IntentReceiver* wertet die dem Intent angehängten Daten aus und implementiert die zur Kommunikation mit der Steckdose benötigten Methoden.

Nachdem mithilfe der *getExtras()*-Methode der String ausgelesen wurde, kann mittels einer simplen if-Abfrage entschieden werden, welche Aktion durchgeführt werden soll.

Die Kommunikation über IPv6-Multicast soll an dieser Stelle anhand der *steckdoseAn()*-Methode (siehe Listing 22) beispielhaft erklärt werden.

Zunächst wird ein neuer Multicast-Socket mit dem Port „4321“ erstellt. Da es bei IPv6 keine Broadcast-Adresse mehr gibt, muss an dieser Stelle mit dem Befehl *joinGroup* der Multicast-Gruppe „ff02::1“ beigetreten werden. Anschließend wird ein String, der den Befehl zum Schalten der Steckdose enthält, in ein Datagramm-Paket verpackt und über den Socket rausgeschickt.

Alle Geräte der WieDAS-Musterwohnung bekommen nun die Nachricht „p018:sd:0001:einschalten“, die Steckdose erkennt den an sie gerichteten Befehlt und führt diesen aus.

```
public void steckdoseAn()
	{
		try {
			socket = new MulticastSocket(4321);
			socket.joinGroup(Inet6Address.getByName("ff02::1"));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		  String msg = "p018:sd:0001:einschalten";
		  byte[] data = msg.getBytes();
		  packet = new DatagramPacket(data, data.length);
		  try {
			packet.setAddress(InetAddress.getBy-
Name("ff02::1"));
			packet.setPort(4321);
			socket.send(packet);
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		  socket.close();
	}
```

Listing 22: Kommunikation mit der Steckdose über IPv6-Multicast

Nach dem gleichen Prinzip können nun auch andere Geräte angesprochen werden.

5.4 Ausnahmefall Türüberwachung

Im Gegensatz zu allen anderen WieDAS-Geräten, wird die Türüberwachung respektive Türsteuerung nicht über IPv6-Multicast, sondern mittels CGI-Befehlen über das Aufrufen einer IPv4-Adresse angesprochen. Da ein Umweg über das OSGi-Framework für diesen Anwendungsfall keinen großen Vorteil bietet, wurde die Funktionalität direkt in die neue WieDAS-Anwendung integriert.

5.4.1 Authentifikation

Da der Zugriff auf die Kamera durch einen Benutzernamen und ein Passwort geschützt ist, muss sich das Smartphone bei jedem Zugriff authentifizieren.

Hierzu werden die Bibliotheken *java.net.Authenticator* und *java.net.PasswordAuthentication* benötigt. Diese werden von der *WebCamAuth*-Klasse verwendet, welche dazu dient, festzustellen, welche Art der Authentifikation der Zugriff erfordert und die nötigen Zugangsdaten zu liefern.

```
import java.net.Authenticator;
import java.net.PasswordAuthentication;
//**Klasse zur Authentifizierung bei der Webcam**//
public class WebCamAuth extends Authenticator
{
   String username = "geheim";
   String password = "geheim";

   public WebCamAuth (String username, String password)
   {
          this.username = username;
          this.password = password;
   }

   @Override
   protected PasswordAuthentication getPasswordAuthentication()
   {
          return new PasswordAuthentication(username, password.toCha-
rArray());
   }
}
```

Listing 23: WebCamAuth-Klasse

Für den Authentisierungsvorgang wird vor jedem Verbindungsaufbau eine Instanz dieser Klasse mithilfe der *setDefault()*-Methode an den *Authenticator* übergeben.

```
Authenticator.setDefault(new WebCamAuth(„geheim“,“geheim“));
```

Listing 24: Authentifikation vor einem Verbindungsaufbau

5.4.2 Kamerabild anzeigen

Über den Aufruf einer festgelegten URL ist es möglich ein Bild von der Kamera zu bekommen. Um einen „Live-Stream“ zu erzeugen, wurde das Intervall für die Wiederholung der *updateUI()*-Methode des *Tuerkontrolle_Fragments* auf 800 Millisekunden verkürzt. Die Methode selbst musste komplett

umgeschrieben werden. Das Layout der Informationsseite der Türüberwachung hat eine ImageView bekommen, welche von der *updateUI()*-Methode aktualisiert wird.

```
public void updateUI(){

	tb = (ImageView) getView().findViewById(R.id.tuerBild);
	if(startcam == 0)
	{
		webcam = new WebCamTask();
		webcam.execute();
		startcam = 1;
	}
	Drawable drawable = Drawable.createFromStream(inputStream, null);
	tb.setImageDrawable(drawable);

}
```

Listing 25: updateUI()-Methode des Tuerkontrolle_Fragments

Damit die WieDAS-App den Design-Richtlinien für Anwendungen für Android 4.x genügt, übernimmt ein *WebCamTask*, welcher beim ersten Aufruf der Methode gestartet wird, das eigentliche Laden des Bildes. Mithilfe einer if-Abfrage wird geprüft, ob der Task bereits ausgeführt wird. Anschließend wird eine Drawable-Instanz aus dem vom *WebCamTask* erzeugten inputStream angelegt und an die ImageView des Fragments übergeben.

Die *WebCamTask*-Klasse wird in Listing 26 gezeigt und ist simpel gehalten. In ihrer *doInBackground()*-Methode wird nach der Authentifikation eine Verbindung zur Kamera aufgebaut und mittels URL-Aufruf ein Bild angefordert. Dieses wird anschließend durch den Aufruf der *getContent()*-Methode in einen inputStream geschrieben, welcher an das *Tuerkontrolle_Fragment* weitergereicht wird.

Abbildung 11: Informationsseite Türüberwachung

```
protected Void doInBackground(Void... params) {
// TODO Auto-generated method stub
       while(true)
       {
         try {

                Authenticator.setDefault(new WebCamAuth("geheim", "ge-
heim"));
                myUrl = new URL("http://XX.XX.XX.XX/image/jpeg.cgi");

               }
          catch (MalformedURLException e) {
               // TODO Auto-generated catch block
                 e.printStackTrace();
               }

                 try {
                       inputStream = (InputStream)myUrl.getContent();
                       Tuerkontrolle_Fragment.inputStream = inputStream;
                      }
                  catch (IOException e) {
                      // TODO Auto-generated catch block
                       e.printStackTrace();
                      }
                   }

       }
```

Listing 26: doInBackground()-Methode des WebCamTask

5.4.3 Türöffner bedienen

Der Türöffner kann ebenfalls über den Aufruf einer URL bedient werden. In der WieDAS-App wurden für diesen Zweck – analog zur im vorangegangenen Kapitel vorgestellten Lösung – ein *TuerOeffnerTask* sowie ein *TuerSchliesserTask* erstellt, welche bei der Ereignisverarbeitung der Informationsseite gestartet werden.

Innerhalb der *doInBackground()*-Methode des *TuerOeffnerTasks* wird ein neuer Thread erstellt, der die URLs zum Öffnen und Schließen der Tür mit einer zeitlichen Verzögerung nacheinander aufrufen soll. In der *run()*-Methode des Threads wird zuerst die URL zum Öffnen der Tür aufgerufen. Danach wird der Thread mithilfe der *sleep()*-Methode für 2 Sekunden pausiert und ruft anschließend die URL zum Schließen der Tür auf. Nach erfolgreicher Ausführung beendet sich der Thread automatisch. Die Implementierung wird in Listing 27 gezeigt.

```
protected Void doInBackground(Void... arg0) {
                // TODO Auto-generated method stub
                Thread thread = new Thread()
                    {
                        @Override
                        public void run() {
                            try {
                                threadLock = 0;
                                   while(threadLock == 0)
                                   {
                                   try {

                                                Authenticator.setDefault(new
WebCamAuth("geheim", "geheim"));
                                                myUrl = new
URL("http://XX.XX.XX.XX/dev/gpioCtrl.cgi?out1=on");

                                        } catch (MalformedURLException e) {
                                                // TODO Auto-generated catch block
                                                e.printStackTrace();
                                        }

                                        try {
                                                inputStream = (In-
putStream)myUrl.getContent();
                                        } catch (IOException e) {
                                                // TODO Auto-generated catch block
                                                e.printStackTrace();
                                        }
                                     sleep(2000);

                                     try {

                                                        Authenticator.setDefault(new
WebCamAuth("geheim", "geheim"));
                                                        myUrl = new
URL("http://XX.XX.XX.XX/dev/gpioCtrl.cgi?out1=off");
                                                        threadLock = 1;

                                                } catch (MalformedURLException e) {
                                                        // TODO Auto-generated catch
block
                                                        e.printStackTrace();
                                                }

                                                try {
                                                        inputStream = (In-
putStream)myUrl.getContent();
                                                } catch (IOException e) {
                                                        // TODO Auto-generated catch
block
                                                        e.printStackTrace();
                                                }
                                    }
                                }
                                 catch (InterruptedException e) {
                                    e.printStackTrace();
                                }
                    }

                    };

                    thread.start();
                return null;
        }
```

Listing 27: doInBackground()-Methode des TuerOeffnerTask

6 Fazit und Ausblick

Im Rahmen dieses Praxisprojekts wurden erste Erfahrungen mit der OSGi- sowie Android-Programmierung gemacht. Trotz fehlender Vorkenntnisse in diesen Bereichen, konnte das Projekt dank guter Dokumentation vorangegangener Arbeiten und des Developer-Tutorials [4] zufriedenstellend und in der gegebenen Zeit umgesetzt werden.

Realisiert wurde eine Android-Anwendung, mit der es möglich ist, vier von sieben Geräten der WieDAS-Musterwohnung zu überwachen und zu bedienen. Die restlichen drei können mithilfe des nun vorliegenden Quellcodes und der Dokumentation mit geringem Aufwand eingebunden werden.

Probleme tauchten auf bei der Anpassung des Felix-Frameworks an die Design-Richtlinien neuer Android-Versionen, welche auch bei der WieDAS-App-Programmierung beachtet werden mussten. Die Lösung wurde umgesetzt und dokumentiert.

Das hier vorgestellte System funktioniert zur Zeit nur innerhalb des WLANs, in welchem sich die Geräte befinden. Interessant wäre eine Lösung für den Zugriff über UMTS respektive von einem anderen Netzwerk aus. Eine Lösung hierfür zu finden wäre ein mögliches Thema für eine Bachelor-Thesis.

Literaturverzeichnis

[1] WieDAS Projekt Team. WieDAS Projekt Website. 2013.

http://www.wiedas.org

[2] Schmitz, Thomas. Entwicklung einer mobilen Software zum Steuern und Überwachen von Wohnungstüren auf Basis von Android im Umfeld von Ambient Assisted Living. 2011.

[3] Wientzek, Niels. Konzeption und Implementierung eines Android-Services für das OSGi-Framework. 2012.

[4] Google Inc. Android Developers. 2013

http://developer.android.com

[5] The Apache Software Foundation. Apache Felix Framework. 2013.

http://felix.apache.org/site/index.html

[6] Schmitz, Thomas. Entwicklung einer OSGI - Service - Komponente zum dynamischen Laden von Benutzeroberflächen für Android im Umfeld von AAL. 2011.

Abbildungsverzeichnis

Listingverzeichnis

Listingverzeichnis